3 Musketeers - Battlecode 2021 Post Mortem

Winston Cheung, Maxwell Jones, David Lyons, Bharath Sreenivas

January 2021

1 Introduction

1.1 Battlecode Introduction

Battlecode is an MIT AI competition run every year throughout the month of January. At the beginning of the month, they release a new 2 player game, at which point teams have to code up a bot that plays said game. There are multiple Tournaments throughout the month, ending in a tournament for the top 16 teams out of the hundreds that submit code for cash prizes.

1.2 Team Introduction

We're four computer science students in our Sophomore year at Carnegie Mellon University. We origionally chose the name **3 Musketeers** because there was only 3 of us, but some time during the first week we added another member, so we ended up with 4 people for most of the tournament. It was our first time doing Battlecode, and because of that we didn't really know what we were doing for a large part of the process.

As the month progressed and we adapted to the changing meta, we ended up qualifying for the final tournament and placed 9th out of the 16 teams. Given our inexperience, we were pretty happy with that result, and are hoping to do even better in the coming years. This is a newbie perspective on making a deep run in the tournament, with highlights of our

strategy and what we learned as the month progressed. If you want to take a look at our code, it is linked here, and our final player is under src/musketeerplayerfinal.

2 Game Overview

There were two methods to win a game in Battlecode 2021:

- completely destroy your opponent by capturing or destroying all of their units
- survive until the end of the game (some set number of rounds) and have more votes than your opponent at the end[1]

Games are played on a 2D rectangluar maps with widths and heights between 32 and 64 tiles each. Each tile has a certain passability from .1 to 1, which indicates how fast robots can move on these tiles. Lower passability means that robots will take longer to move. For instance, it would take a robot 10 turns to move through a tile with .1 passability[2].

All bots also had 24 bit flags, used to communicate with each other. There were a total of 4 different types of units that each team could control, each with its own specific function.

- **Enlightenment Centers (ECs):** ECs are the most important unit of the game with each team starting out with a set amount of ECs on the map. Each EC has some amount of influence (effectively money) that it can use to produce the other robot types, with a cooldown of 1 turn between production of new units.

  They gain influence passively each turn, dictated by a set function based on the round number[3]. ECs have large vision, but cannot move. It is also important to note that ECs can be converted from one team to another, or even can start out neutral, doing nothing, until one team finds them and converts them (more on this later).

- **Politicians:** Politicians are one of the two main attacking bots in the game. They start with some amount of conviction and influence, with conviction basically referring to how much oomph they hold at a given time. Politicians can empower within a certain radius, and when they do this, their conviction is split equally among all surrounding towers in that radius, whether friend or enemy, after a tax of 10 conviction. If enemy politicians or ECs go into a negative amount of influence, they are converted to your side. Politicians can also convert neutral ECs to your side.

- **Slanderers:** Slanderers are the main money makers in the game. when spawned, they make some amount of money per turn. They keep making money for 50 turns, and are converted into politicians with equal conviction/influence after 300 turns. Slanderers have the smallest vision and cannot attack, meaning that it is increasingly important to try to protect them to make your ECs as much money as possible.

[1] more on this later

[2] As it would turn out, different maps would drastically change optimal strategy for bots, and so having a bot that was able to adapt to different maps/different positions withing maps was key to overall success

[3] The later the round, the more passive influence an EC gains

- **Muckrakers:** Muckrakers are the second type of attack bot in the game. They are slower than politicians, but with larger vision/attack range. While muckrakers cannot empower, they are able to expose slanderers, killing them in one round regardless of the influence of the slanderer getting killed or the muckraker doing the killing. Exposing slanderers would yield a temporary buff to politician attacks, proportional to the influence of the slanderer. They cannot be converted from team to team, but instead die if they get put into negative conviction by an enemy politician.

NOTE ABOUT FLAGS: ECs can see the flags of all units regardless of distance. Non-EC units can see the flags of all ECs as well as all flags in their sensing radius. in order for ECs to see flags of other ECs, they need the robotID of other ECs, which they do not have at the beginning of the game.

3 Bot Development and Meta

3.1 Release to Sprint 1

3.1.1 EC States

Before we could begin true development of these strategies, we wanted to nail down solid fundamental code that we could build off of. The main structural support of our code was our control of the specific state of our enlightenment centers. we had an enum of various states that our ECs could potentially be in, and we switched between states whenever we deemed necessary. In order to keep track of what triggered our entrance into a specific state, we made use of a stack, dubbed the stateStack. One example use case was when we were building a politician to attack an enemy base. We set up a state called saving for rush, from which we entered a rushing state when we had enough money to build our rush politician. We needed to store the state that we were in before saving for rush, so that after we build our rush politician (which took higher priority than everything else), we could pop off the state stack and resume previous duties. This was helpful for the future, so we could simply add more states and make modifications easily.

3.1.2 Communications

Comms is a big part of the game, but with only 24 bits to store information, we had to be clever to bit-pack our information so our units could read all the information they needed. For example, when a unit found an enemy or neutral EC, they broadcasted the EC location using our flag structure. In our comms class, we set up InformationCategory as an enum, where we described what flag type was being sent so units knew how to decipher the bits. When a unit sends a location back to its home EC, it uses the NEUTRAL_EC or ENEMY_EC information category, telling its home which EC type it found. In addition, we use dx and dy from home as a location mechanism, storing dx in 7 bits and dy in another 7 bits. We also stored a logarithm of the ECs influence, so we knew how big our rush politician would have to be. These values were constantly getting updated at the home EC, as more units broadcast the location and updated influence values of enemy ECs.

We also use flags when we spawn robots, indicating the robot type that will be created. Since we have various politician types, for example, we sometimes want to build an explorer, and we sometimes want to build a rush politician. In this case, the EC can set a flag to tell the RobotPlayer, which handles spawning robots, which robot type we want to create.

3.1.3 General Initial Strategy

As the game was released, we spent a long time reading the rules and asking questions in the discord, eager to understand what competitors' perspectives on strategies would be. The first thing we noticed by chance was that muckrakers could box an enemy in, preventing them from building units and gaining income as a result. On the other hand, in order to make income ourselves and avoid the enemy from gaining a big buff, we wanted to make slanderers and protect them as well. We rolled with this strategy for the beginning of the tournament. The main aspects of this strategy involved two things: bodyguarding our slanderers, and rushing muckrakers.

- **Bodyguards:** When a slanderer was spawned, we wanted to send a politician to follow it. These politicians followed a slanderer, and when they get near a slanderer, they do not move away from it. As a result, they can empower on enemy muckrakers that came by to expose our slanderers. This was an interesting strategy that worked in the beginning of the tournament.

- **Muck Rush:** We spawned many muckrakers with the intent of exploring the map. Originally, we thought that a small muckraker would not be able to survive the trip across the map and land near an enemy EC, so we spawned 50-influence muckrakers at the start of the tournament. Eventually, we realized that increasing our number of muckrakers spawned would allow us to overwhelm the enemy regardless of how big the muckrakers were, so we switched to spawning small 1-influence muckrakers. When a muckraker found an enemy EC, it simply sat there, hoping to get some neighboring muckrakers to build a wall around the enemy EC, blocking it in. We built so many of these muckrakers that eventually found an enemy EC, making it hard for enemies to advance. One way we defended against this same attack was a state called REMOVING_BLOCKAGE, where if we have more than 6 enemies in the surrounding squares of our EC, we build a politician that immediately empowers, in order to clear the build spots for our EC.

In general, our units moved away from enemies that could attack them, which proved a decent initial movement strategy for us. In addition, when we rushed a base and it was already taken over, our failed rush politicians (which had a significant amount of influence), would turn into what we called Golem Politicians. These politicians would patrol the base they were assigned, waiting to kill big enemies that approached it. This was a nice defensive strategy in the beginning, but we eventually got rid of it as the meta evolved.

3.1.4 Self-Empowering

A big development we found in the rules was that when a politician empowers, it imparts its conviction even on friendly units and buildings. Since ECs had conviction equal to influence at all times, we exploited this feature whenever the buff was above 5. If enough enemy slanderers were killed and the buff was above 5, we spawned a large politician (roughly 3/4 of our influence) to immediately empower, giving us 4x returns on the influence we used to create the politician. This allowed us to exponentially grow our influence in the few rounds that the buff was high during a game. This gave us a significant influence advantage a lot of the time, and allowed us to stay ahead. However, in an attempt to nerf this, **Teh Devs** made the initial cooldown of politicians and muckrakers to be 10 turns. We removed self-empowering for a brief period, before quickly putting it back in after some scrims with **Kryptonite**, seeing that they were still using it effectively.

3.1.5 Bytecode Optimizations

We realized that storing the IDs of the units that an EC created would be crucial, as we wanted to loop through them every turn to update our information about enemy and other friendly units. We stored these IDs in an arraylist originally, but Java.util data structures proved to be very costly in terms of bytecode. As a result, we developed a custom data structure called a FastIterableIntSet, which used a StringBuilder to store a set separated by the "^" character. All operations on StringBuilders conveniently only cost 1 bytecode, which allowed us to quickly iterate and update the size of them. This find was key as we started to spawn more units in later stages of the game. We used this logic for future iterations of our bot to build fast maps, and sets that stored information other than integers.

3.2 Sprint 1 to Sprint 2

3.2.1 Formations

After sprint 1, the primary thing we saw from other teams that we wanted to implement was formations. Whereas our bots were very individual and did what was best for them, the top teams had coordinated groups of units, so we started to do the same. The first change was to keep slanderers close to the EC. As such, we changed our code and removed bodyguards entirely, instead creating a new class of politicians called protectors that rotated around the base to protect our slanderers. These politicians would attempt to push enemies away from the EC so that the enemies could never find us, and if enemies got too close, the politicians would empower. Moreover, rather than just having slanderers move randomly and avoid enemies, we programmed the slanderers to have an intentional formation where they all stay exactly one diagonal square from the nearest slanderer. This was useful because it allows units to move between these diagonal lines and escape the EC, whereas before the slanderers could become cluttered and prevent troop movement.

Lattices also became rather popular, spreading units out evenly across the entire map so that the opponent couldn't get anywhere without you knowing. We wanted to have both our protector politicians and our muckrakers to lattice, and only with their respective types. We spent a lot of time tweaking and trying different heuristics that would achieve the best

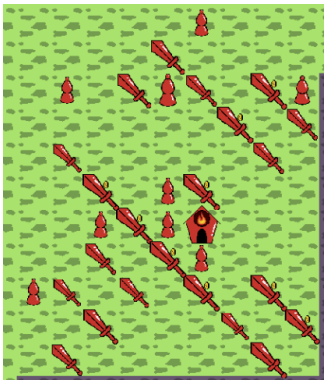


Figure 1: Slanderers in diagonal lines to allow troop movement

lattice, eventually settling on just moving away from the closest unit of the same type, which worked moderately well. Every team seemed to have different methods of achieving their lattice, but we wanna give special props to team **monky**, who created a beautiful and amazingly effective lattice by making their units behave like repelling point charges.

3.2.2 Enemy Reporting

In addition to improved formations making our bots look more like an actual team, we also greatly improved our communications code. Namely, we had all our units report anytime they find an enemy, and we calculated the average enemy direction. This was important, because it allowed us to do two things. First, all our slanderers, in groups close to the EC, would move away from the average enemy direction. We would also spawn them away from the average enemy direction. Second, we created a new class of muckraker called hunter muckers. These muckrakers, when they find enemy ECs, charge them. If we already know where an enemy EC is, we give the hunter mucker this location when we spawn it. If someone else finds an enemy EC and reports it, these hunter muckers answer the "attack call" and can swarm enemy bases, preventing them from creating units. This led to one small problem, though, which was that our rush politicians would have trouble destroying enemy ECs if the EC was surrounded by our hunter muckers. To fix this, we implemented a tactic called muckraker dispersion, where muckrakers move away from politicians when they read the flag and see that it is rushing a base. This would effectively create a tunnel for the rush politician to enter, uninterrupted by the swarm of muckrakers. Then, once they finish rushing the base, the muckrakers could return to their position.

3.2.3 Control Flow

During testing, we realized that we weren't consistent throughout the code; sometimes we'd switch states at the beginning, and sometimes we'd switch at the end. For example, before the switch statement, we had a function called toggleBuildingProtectors that either starts building those circular rotating protectors or stops. Then, inside the switch statement, in the BUILDING_PROTECTORS case, we go into state RUSHING if we're ready. First of all, this process could repeat indefinitely, skipping turns and creating state stacks of sizes

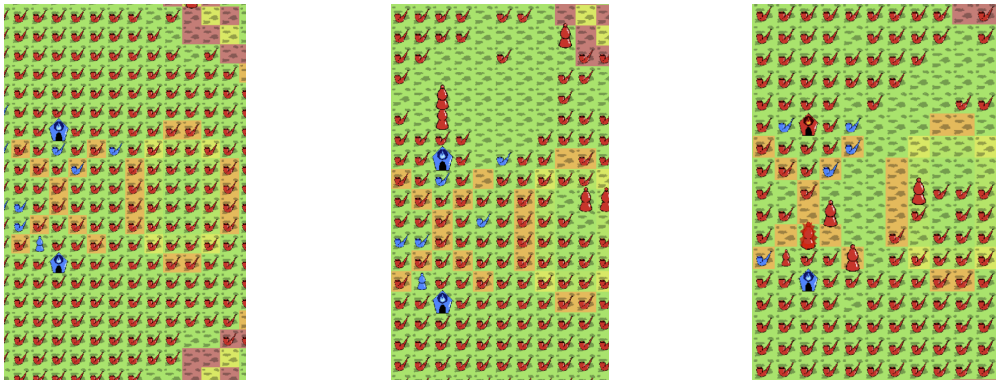


Figure 2: Muckraker dispersion creating a tunnel to the enemy ECs

>100. Second of all, constantly switching states is prone to errors. To fix this, we overhauled the entirety of our EC control flow. We took out all state switching from inside the switch statement and had one big nested if statement that had a set of prioritized things: Can we do our first priority? If so, great. Go into that state and enter the switch case. If not, move on to priority number 2. First of all, this avoids large state stacks. Since you can only move upwards in terms of priority, the state stack, rather than having a maximum size of several hundred, had a maximum size of about 11, since all that could be stored is lower priorities, in order. Second of all, it ensures that we don't waste turns. We decide on a state at the beginning and then execute, never turning back. Third of all, we ensure that we only toggle states *before* we do anything in that state for that turn, which is way less prone to errors where robots think they're in the wrong state.[4]

3.2.4 Advanced Bidding

We wanted a bidding strategy that won a majority of the bids without causing us to sacrifice more influence than we needed to. Our original thought was to bid based on what phase we were in or what kind of unit we were creating, but eventually we came up with a technique similar to binary search, where we have a high bid and low bid. If we win a bid, we lower the high bid to our previous bid. If we lose a bid, we raise the low bid to our previous bid. When necessary, we half the lower bid or double the higher bid. Eventually, as ECs reach an equilibrium influence, we begin to narrow down on the opponents' average bid, allowing us to win the majority of the bids whilst only bidding about one or two more than our opponents. This allowed us to win a lot more bids than we usually do, but it also allowed us to be more effective in our takeovers, since we had more influence. Later, we improved this bidding strategy by adding an equilibrium feature. If bidding X results in a win, and bidding X-1 results in a loss, we continue bidding X until we lose.

[4]In hindsight, we could have avoided these nested ifs by putting control flow decisions in a method and just returning when we'd decided to change states.

3.3 Sprint 2 to Quals

3.3.1 EC Restructure

We noticed that a lot of teams were hardcoding their initial moves for the first ∼30 robots, so we decided to do that too. While we tweaked our initial robots a bit, the main idea was first make a single large slanderer to make money, then send out 8 scout muckrakers in the cardinal directions, along with 2 explorer politicians to explore the map and find enemies[5]. After that, we would alternate between slanderers and politicians, keeping the ratio at about 2 politicians per slanderer[6]

After hardcoding the first part of the game, we moved on to two main states, CHILLING, and ACCELERATED_SLANDERERS.

- **CHILLING** was what to do when you don't really have anything going on, but there are some enemies near the base. In this state we built a ratio of 2 politicians to 2 muckrakers to 1 slanderer, but only made slanderers of size over 100[7]. If we didn't have enough money to build a slanderer, we would simply build more 1 influence muckrakers until we did.
- **ACCELERATED_SLANDERERS** was what to do when there are no enemies near. In this case, we only build slanderers and politicians, with 2 politicians per slanderer[8]

We could still rush enemy/neutral bases from either of these states, so the state stack was still in effect, but adding these two main states to our EC code and hardcoding our initial strategy greatly improved our bot's performance.

3.3.2 Buff Mucks

Something that we noticed early on is that a lot of the top teams made slanderers very early on, and didn't protect them at the beginning of the game, as they would kill any enemy muckraker before they were in sensing radius of the base. In order to take advantage of this, we decided to send a buff Muck (influence 150 muckraker[9]), to the first location that one of our original muckrakers from the hard coded first moves died. This was likely to be very close to sensing radius of the enemy base, and the goal was that this buff muckraker would expose some slanderers before it's eventual death. This ended up working very well for us and made our bot much more deadly. In addition, we decided to send buff mucks(size 150) to any existing enemy bases that we knew about to make sure that slanderer production was always low.

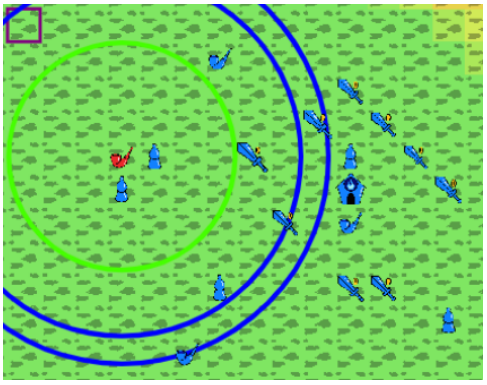
[5]politicians are faster than muckrakers, so they can explore faster, but muckrakers are more likely to get killed near an enemy base, tipping us off as to where enemy bases may be before we ever lay eyes on them

[6]this early game was later changed to 1 politician per slanderer in order to make more money

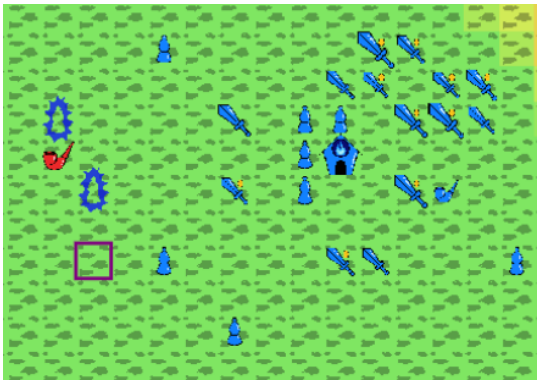
[7]this was changed later

[8]we probably should have made more slanderers in this mode, but oh well

[9]At this point, we were one of the only top teams to NOT use buff mucks, so we were kind of late to the party adding them in.



(a) Scout Muckraker dying while outside of sensing radius of EC



(b) Buff Muckraker going to Scout location death, unable to be killed easily en route to slanderers

3.3.3 About to Die

In the sprint 2 tournament, one thing we noticed was that other teams would make 1 cost muckrakers to suround their base when a large politician was coming to overtake it. This would force the large politician to split its empower influence between the base and the 1 cost muckrakers, possibly saving the base from dying. We added ABOUT_TO_DIE mode in order to do this as well if we see a politician bigger to overtake the base coming for us.

3.4 Post-Quals to Finals

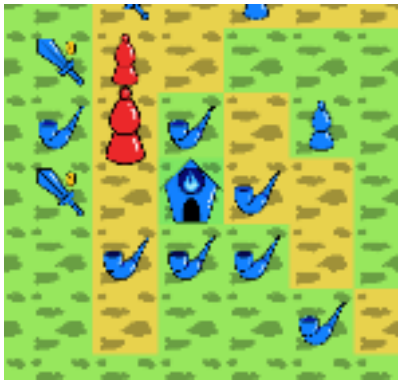
Self-empowering was completely removed just days before quals, a huge game change that teams didn't have much time to adapt for in time for quals. As a result, no new meta had developed for it and nobody really knew what to expect as we got closer to finals.

Nonetheless, the one thing we knew we needed to improve on going into finals was using our influence more efficiently. With that in mind, we focused mostly on improving offense and tweaking micro and defense, keeping our economy the same. This would later come back to bite us, as almost every finals map rewarded going super hard on eco and killed any bot that relied on rushing early to mid-game.

3.4.1 Rush Buddies

With other teams also aiming to dilute the attacks of our own rush politicians, we needed some way to combat this ourselves. Our plan of attack[10] was to send a supporting politician that would empower before the larger, "head" politician, clearing out any smaller units. The key to this strategy working relied on turn order. Because robots get their turns in the order they are created, if we built these politicians one right after the other, we could (almost) guarantee that they would get their turns in succession and that the attack would be successful.

[10]Pun *intended*



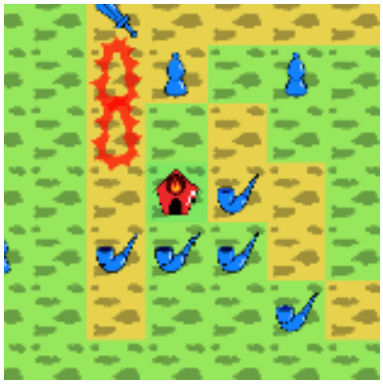


Figure 4: Rush buddy system successfully converting an enemy EC

3.4.2 Other Changes

On the defensive side, we started communicating the influence of buff mucks in order to build a single politician just large enough to kill it. This would help us avoid the problem of multiple empower taxes being paid if we were to simply use our smaller politicians to defend.

Most movement micro didn't change, although we specifically wanted our politicians to get better unit/influence trades when defending. We had a big problem of politicians far away from enemies empowering, resulting in effectively no damage done to the enemy due to dilution. Like most teams, we ended up deciding to loop through attack radii to determine the one that would do the most damage or kill the most enemies.

The final change we made was allowing ECs to check on each other when taken over. If we knew that one of our friendly ECs got captured, we would send a scout there that would report back if we needed to send a big rush politician to recapture, or if the friendly EC had units that handled it and already recaptured it.

4 Thoughts about the Game

Since this was our first ever Battlecode tournament, by default it was our favorite. With this being said however, we all really enjoyed the game and trying to master it to the best of our abilities. While the top teams all did eco pretty hard, there was still a good amount of different strategies employed by different top teams, which is good.

If there was a change that we could make, it would probably be adding more consistency between the maps used in scrims and the maps used in the final tournament, especially considering different maps can drastically change what strategies are optimal (We probably should have made our code more eco centric if we hadn't seen enemies in a while, but we think that the point still holds).

5 Suggestions for New Players

First, we are just going to reference a blurb from 2020 in Java Best Waifu's Post Mortem as it sums up a lot of stuff really well:

> "If I had to say what are the most important factors to consider when building your bot, I would say **Simplicity**, **Robustness** and **Structured Code**. In my personal experience, every time I'd try to implement a sophisticated strategy that requires a lot of coordination it would always flop since everything that can go wrong does go wrong. Usually the bots that perform the best are those that perform the basics really well (navigation, communication, micro, macro, etc.). It is especially important to use the first days to implement good and robust primitives for such basics since they'll be needed regardless of the strategy. It is usually more effective to spend your time on the basics and then figure out the optimal strategy depending on what other players do than trying to devise the optimal strategy from the go.
>
> Also, if the code undergoes drastic changes (for instance because you change the bot's main strategy) I would suggest to do a bot from scratch. It is usually faster than expected and it is way better on the long run."

With this being said, there are some things we would like to add.

Don't be afraid to steal ideas. Since the meta in games changes so much, you pretty much have to steal ideas at some point in order to evolve your bot to the top level of competition for any given week.

Rigorously debug/watch matches. Sometimes errors are really hard to find, so it can be super useful to watch playthroughs of matches either against other teams or against old versions of your bot to see if all intended behavior is happening. We found a lot of bugs by going over a bunch of games frame by frame and seeing what was happening (even in wins).

Don't get discouraged if you haven't made progress in a bit. Sometimes it's hard to know what part of your code is causing it to perform sub-optimally. It's possible that you could be working on the completely wrong section, and missing an area that could cause great improvement in performance. With this being said, if you haven't made progress, it would be wise to either shift focus from what you are doing, or just have a general assessment of how your code differs from others to see if that can help you identify the problem.

Use the Discord. Top teams tend to talk about different ideas in discord and being active there can be a good way to stay in the loop about the current meta.

6 Final Thoughts

Overall, We had tons of fun and are definitely going to come back next year in some form or another. A lot of people helped us out along the way, with special shoutouts to **Monky** for being great friends and **Kryptonite**, **Producing Perfection**, and **Blue Dragon** for being part of the CMU gang!